"Nullification of Unknown Malicious Code Execution with Buffer Overflows"
(AOARD-03-4049)

Driverware IMMUNE

2005/05/01

SciencePark Corporation

SCIENCE PARK

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
27 JUL 2006	**Final Report (Technical)**	**04-11-2003 to 30-07-2005**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Nullification of Unknown Malicious Code Execution with Buffer Overflows	**FA520904P0085**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Yoshiyasu Takefuji**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Keio University,5322 Endo, Fujisawa,Kanagawa 252-8520,JP,252-8520**	8. PERFORMING ORGANIZATION REPORT NUMBER **AOARD-034049**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **The US Resarch Labolatory, AOARD/AFOSR, Unit 45002, APO, AP, 96337-5002**	10. SPONSOR/MONITOR'S ACRONYM(S) **AOARD/AFOSR**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **AOARD-034049**

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The final report describes the working mechanism of ?IMMUNE DRIVER? system. This system uses the functions that are regularly not used by IA32 process which maintains the highest market share at present to interrupt program when buffer overflow is generated by virus program.

15. SUBJECT TERMS
Computer Security Technology, Computer Operating Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**37**	

Contests

1. Summary

This paper describes the working mechanism of IMMUNE DRIVER. This system uses the functions that are regularly not used function by IA32 process which maintains the highest market share at present to interrupt program when buffer overflow generated by virus program.

2. Mandatory Environment

CPU: Intel Pentium Pro or higher processor

*IA-32 Processor with processor instruction trace

OS: Microsoft Windows 2000 SP0

Development environment for IMUUNE system:

Microsoft Visual Studio 6.0 SP4

Compuware DriverStudio 2.7 or later

3. Structure

The following is the sketch of this system

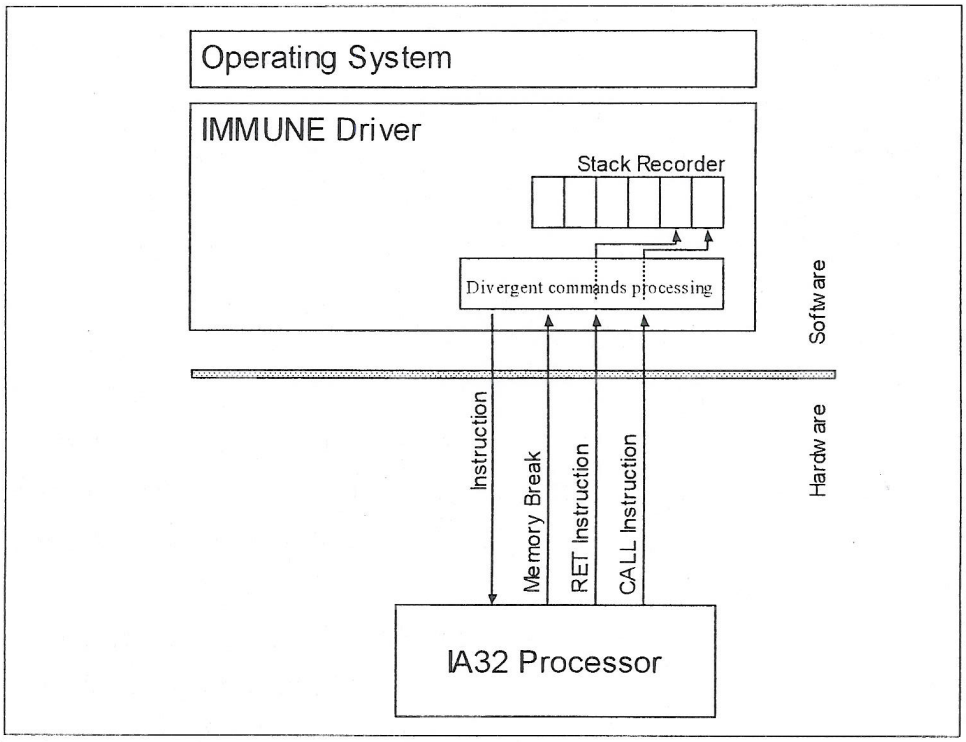


Fig.3-1 Sketch of the system

4. Mechanism

4.1. Brief Flowing Chart

This system consists of five parts: [Initialization], [Process detection], [Event registration], [Instruction trace], and [Process termination].

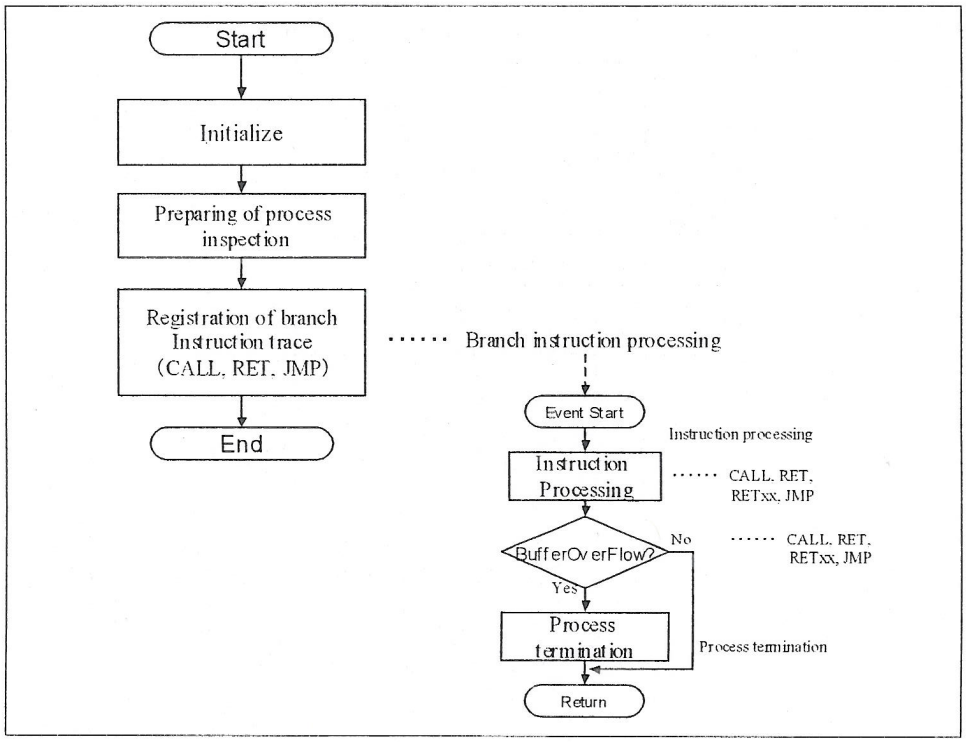


Fig. 4-1 Brief flowing chart of IMMUNE DRIVER

(1) Initialization

The process works on the starting of IMMUNE driver.

(2) Process detection

The preparatory works of detection the buffer overflow before the target process generated.

(3) Registration of branch instruction trace event

Right after the target process generated, the virus execute buffer overflow code by branch instruction. That is why the events of branch instruction should be monitored, of which only the buffer overflow caused by the instruction of CALL/RET/RETxx are to be detected.

(4) Instruction trace

When the registered event occurs, the following methods are to be executed depending on the branch instruction.

① CALL instruction processing: Storing the return address of CALL instruction in our stack recorder.

② RET instruction processing: Compare the return addresses in stack recorder with the current return addresses when CALL instruction executed. If they are not match, it is considered that the return address was changed illegally. This is regarded as buffer overflow, which will forcefully terminate the process.

(5) Process termination

The process that caused buffer overflow is terminated forcibly.

4.2. Initial processing

Flow chart of initialization

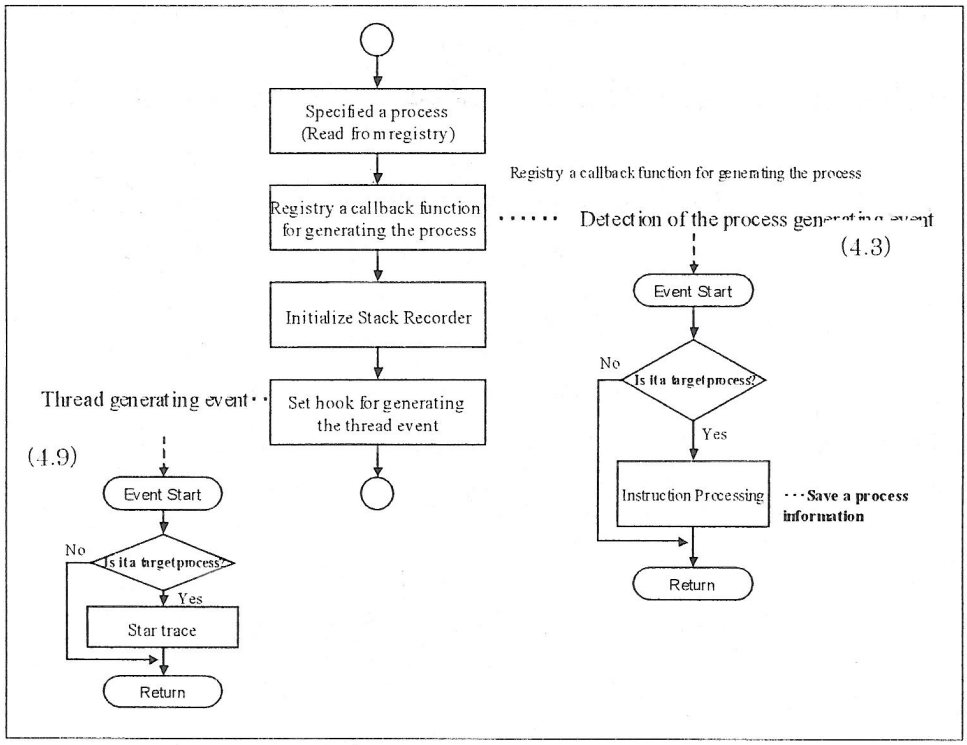


Fig. 4-2 Initial processing

(1) Set process name (by initial parameter)

Specify the process name of monitored buffer overflow from registry.

(2) Register a process generation callback function

To register a process generation callback function, call the PsSetLoadImageNotifyRoutine() function provided by Windows.

Prototype of callback function is defined as following

```
void LoadImageNotifyRoutine(
    PUNICODE_STRING   FullImageName,
    HANDLE   ProcessId,
    PIMAGE_INFO   ImageInfo
    )
```

(3) Initialize Stack recorder

It gives a room and is initialized in the stack recorder. The stack recorder is a buffer used to trace the branch instruction while target process executes. Also, it dynamically gives a room in stack recorder whenever sub thread of the specified process is generated. The stack recorder structure is defined as

```
typedef struct _ASO_STACK_LIST{
    LIST_ENTRY m_ListEntry;
    ULONG ThreadId;
    ULONG *StackPointer;
    LONG CurrentStackLocation;
} ASO_STACK_LIST, *PASO_STACK_LIST;
```

(4) Register thread creation event hooking (suitable for multiple threads)

It is to hook kernel API called NtCreateThread when the sub thread is being generated. For hooking NtCreateThread, entry point of Int2E interrupt handler is replaced with IMMUNE system.

```
IDTR idtr;
PIDTENTRY  OIdt;
PIDTENTRY  NIdt;

__asm   SIDT idtr;

OIdt = (PIDTENTRY)MAKELONG(idtr.LowIDTbase, idtr.HiIDTbase);
gOldINT2EH_Handler = MAKELONG(OIdt[IGATE2E].OffsetLow, OIdt[IGATE2E].OffsetHigh);
NIdt = &(OIdt[IGATE2E]);

__asm {
    CLI
    LEA EAX, ASO_Hook_INT2EH
    MOV EBX, NIdt;
    MOV [EBX], AX
    SHR EAX, 16
    MOV [EBX+6], AX;
    LIDT idtr
    STI
}
```

4.3. Detection of process generation events

Monitoring starts at the beginning of process execution right after being generated. The target process events are got through the following chart:

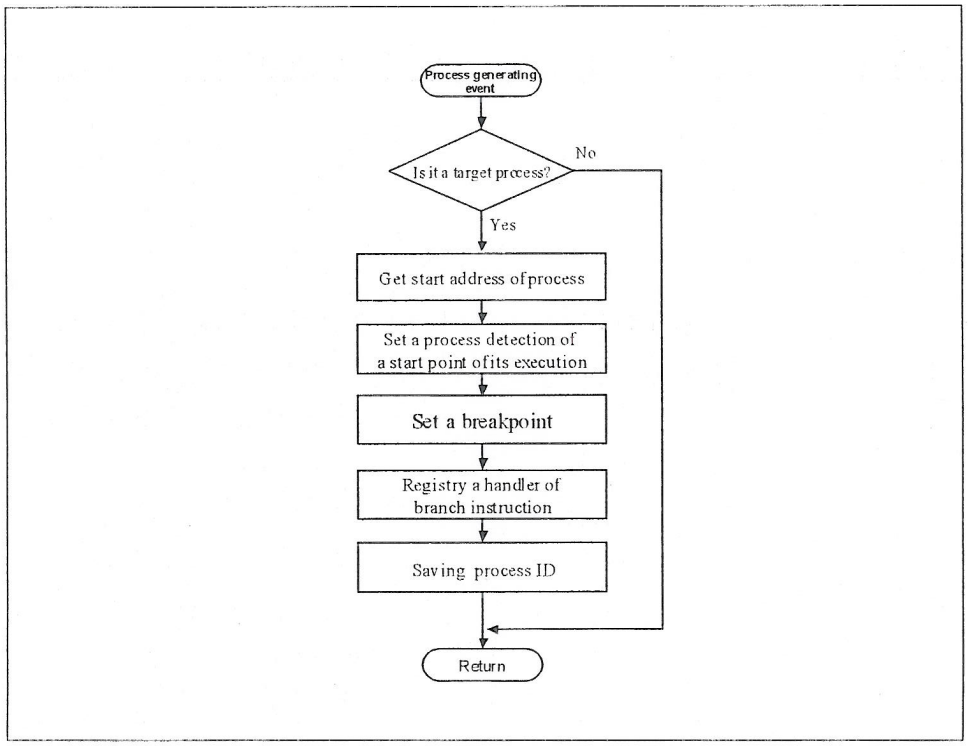


Fig. 4-3 Detection of process generation event

① Detection of process generation

The LoadImageNotifyRoutine is called when any process generates. The function of LoadImageNotifyRoutine is as following

② To determine a target process

The FullImageName (one of the arguments of LoadImageNotifyRoutine) is used to distinguish the target process's module name. If it is believed to be the target process, the process will go as shown in ③.

③ To obtain the process starting address

It gets the program starting address by EXE header the program file. It takes the following codes to get the program starting address.

```
PVOID ImageBase = (PVOID)ImageInfo->ImageBase;

MZ_HEADER *mz_Header = (MZ_HEADER *)ImageBase;
MZ_NE *mz_ne = (MZ_NE *)((char *)ImageBase + sizeof(MZ_HEADER));
IMAGE_NT_HEADERS *ImageNtHeaders =
            (IMAGE_NT_HEADERS *)((char *)ImageBase + mz_ne->ne_header);
char *EntryPoint =
            (char *)((ULONG)ImageInfo->ImageBase+
            ImageNtHeaders->OptionalHeader.AddressOfEntryPoint);
```

④ Setting the detection of process execution starting

Setting a hardware break point in the process starting address got in above mentioned (3). Call the trace callback function (ASO_Hook_INT01H) when the process starts. The following codes are used for the registration

```
MOV EAX, KickStartAddress // Process start address
MOV DR0, EAX

MOV EAX, DR7
OR  EAX, 0x00000000;      // Set LEN0 = 00 (1Byte Length)
OR  EAX, 0x00000000;      // Set R/W0 = 00 (On Execution Only)
OR  EAX, 0x00000200;      // Set GE
OR  EAX, 0x00000002;      // Enable G0
MOV DR7, EAX;             // Set DR7
```

⑤ Setting the execution break point

To set hardware break point in execution entry point to be able to begin the trace as soon as the program starts. Call the INT01 handler when break point is hit during the execution of entry point.

The following codes are used for the registration

```
MOV EAX, KickStartAddress    // Entry Point (main() address)
MOV DR0, EAX

MOV EAX, DR7
OR    EAX, 0x00000000;       // Set LEN0 = 00 (1Byte Length)
OR    EAX, 0x00000000;       // Set R/W0 = 00 (On Execution Only)
OR    EAX, 0x00000200;       // Set GE
OR    EAX, 0x00000002;       // Enable G0
MOV DR7, EAX;                // Set DR7
```

⑥ Registration of branch instruction handler

To register the trace callback function (ASO_Hook_INT01H), Replace the IDT(Interrupt Descriptor Table) of 01H.

The following codes are used for the registration

```
IDTR idtr;
PIDTENTRY Oldt;
PIDTENTRY Nldt;

__asm{
        SIDT idtr;
}

Oldt = (PIDTENTRY)MAKELONG(idtr.LowIDTbase, idtr.HiIDTbase);
gOldINT01H_Handler= MAKELONG(Oldt[IGATE01].OffsetLow, Oldt[IGATE01].OffsetHigh);
Nldt = &(Oldt[IGATE01]);
__asm{
    LEA EAX, ASO_Hook_INT01H// INT01 Hook function
    MOV EBX, Nldt;
    MOV [EBX], AX
    SHR EAX, 16
    MOV [EBX+6], AX;
    LIDT idtr


}
```

⑦ Saving of process ID

When starting, the process gets an ID from Windows. The ID is saved to distinguish the target process when the event occurred by IA-32 branch instruction.

4.4. Starting of tracing

When entry point of program is run, the INT01 handler is called by pre-set break point. From this moment, trace function turns active

The following codes are used for enabling trace function.

```
MOV EAX, DR6;                    // Get DR6 Register Value

// Check MemoryBreak0 Flag
TEST EAX, 0x00000001             // Check MemoryBreak0 bit
JNZ MEMORY_BREAK_BP0
-------------------------------------------------------------------------------
MEMORY_BREAK_BP0:
MOV EAX, DR0
MOV gBreakAddress, EAX

XOR EAX, EAX
MOV DR0, EAX

MOV EAX, DR7
AND  EAX, 0xFFFFFFFD;            // Disable G0
MOV DR7, EAX

MOV EAX, DR6
AND EAX, 0xFFFFFFFE                    // Disable B0
MOV DR6, EAX

JMP MEMORY_BREAK_COMMON
```

4.5. Branch instruction process

When the branch instruction is hit, it is to be processed as below:

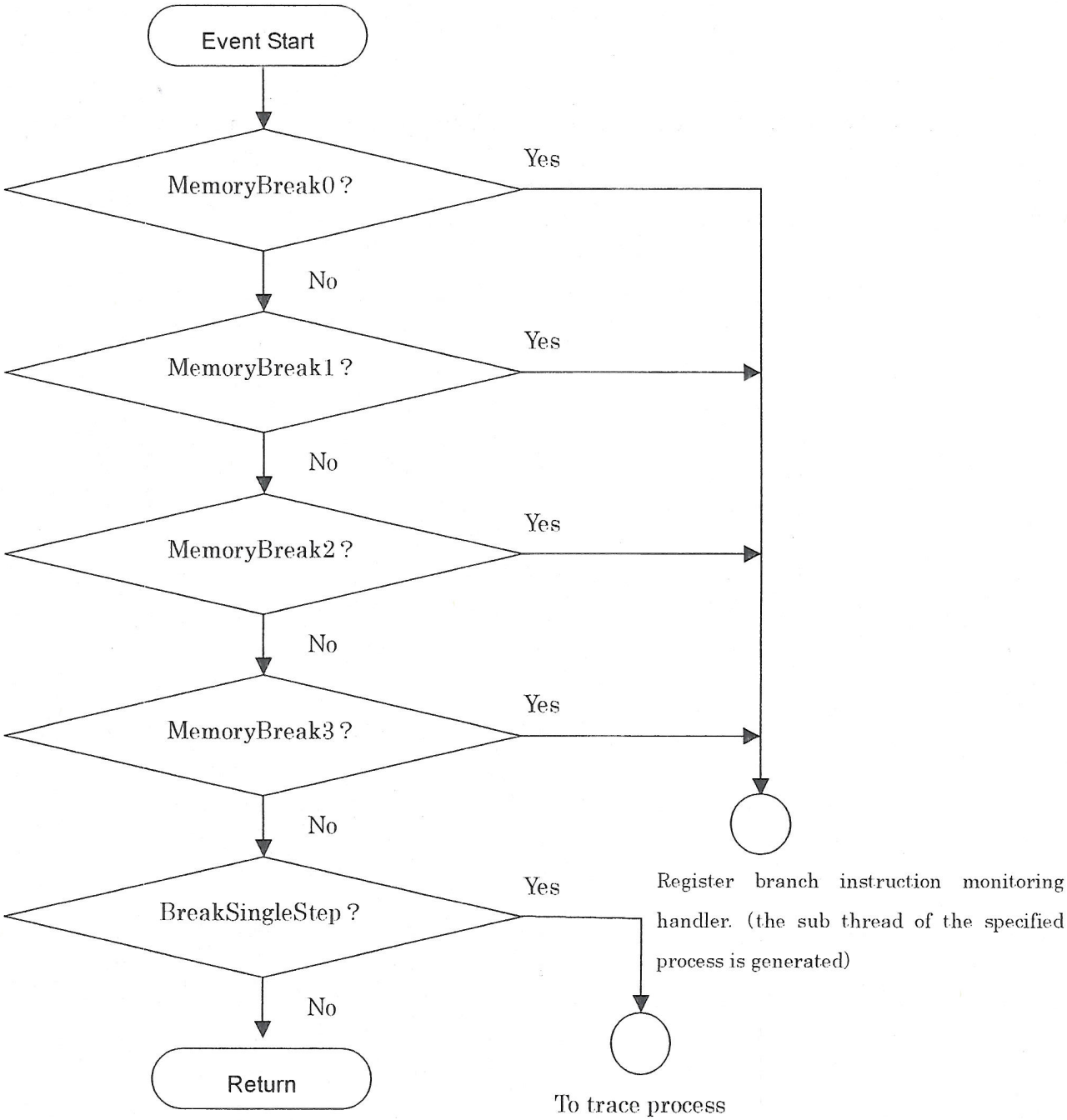
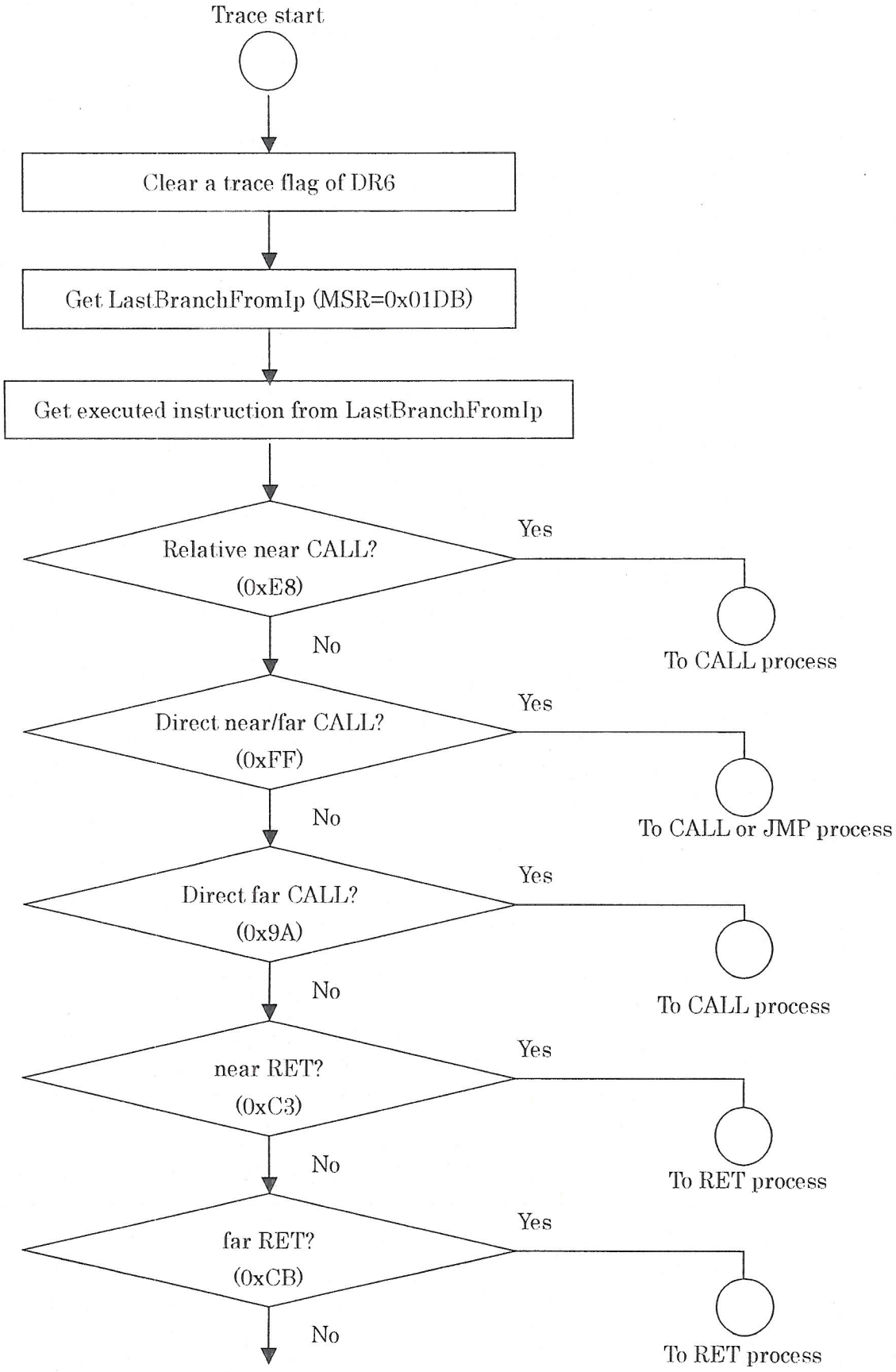
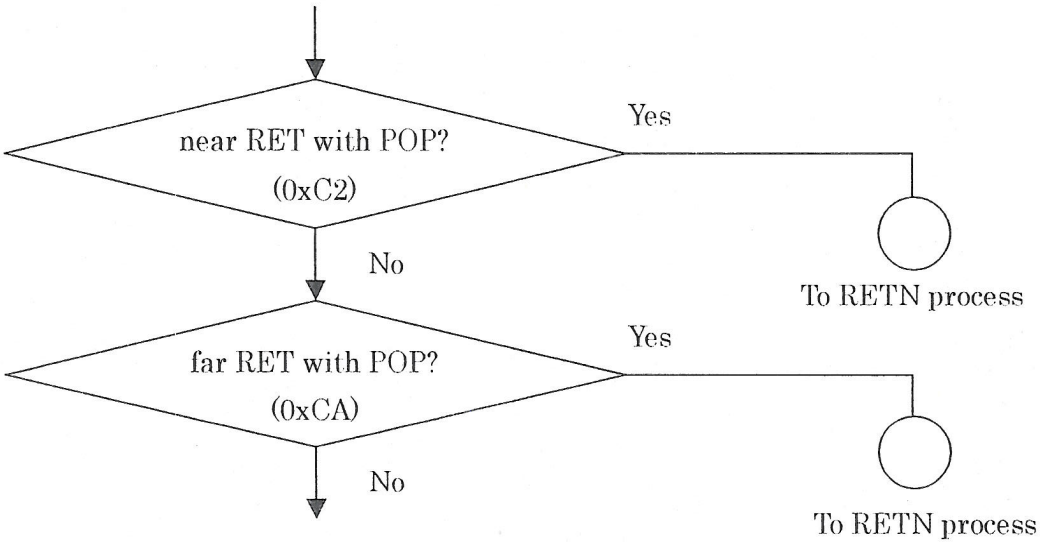
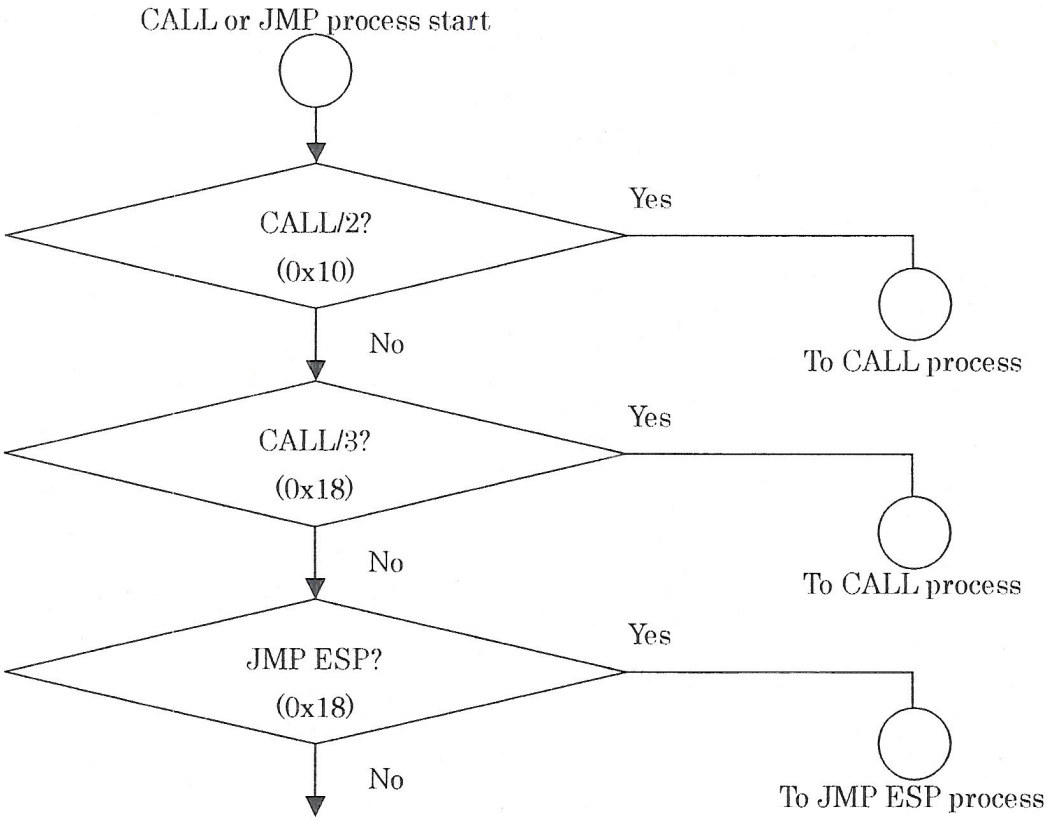


Fig. 4-4


Trace start
Clear a trace flag of DR6
Get LastBranchFromIp (MSR=0x01DB)
Get executed instruction from LastBranchFromIp
Relative near CALL?
(0xE8)
Yes
To CALL process
No
Direct near/far CALL?
(0xFF)
Yes
To CALL or JMP process
No
Direct far CALL?
(0x9A)
Yes
To CALL process
No
near RET?
(0xC3)
Yes
To RET process
No
far RET?
(0xCB)
Yes
To RET process
No


near RET with POP?
(0xC2)
Yes
No
To RETN process
far RET with POP?
(0xCA)
Yes
No
To RETN process


CALL or JMP process start
CALL/2?
(0x10)
Yes
No
To CALL process
CALL/3?
(0x18)
Yes
No
To CALL process
JMP ESP?
(0x18)
Yes
No
To JMP ESP process

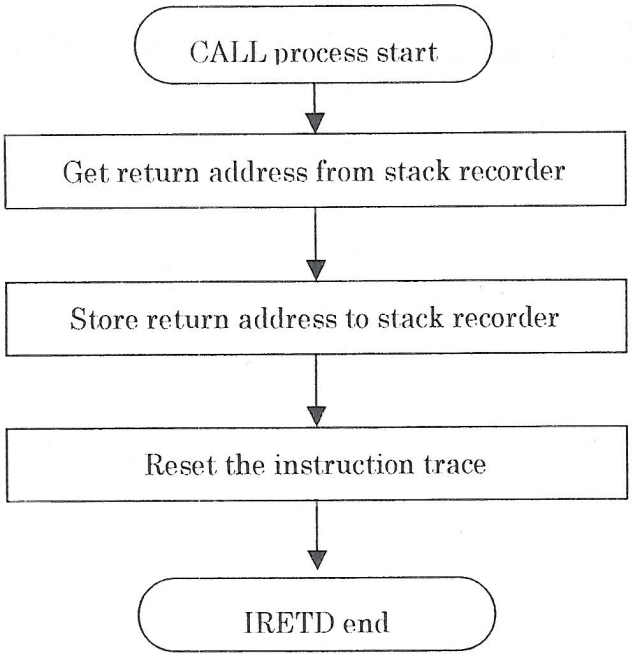
During the branch instruction are executing in process, ASO_Hook_INT01H is called. Only the instruction of CALL and RET that are necessary for IMMUNE system in handler distinguish according the following codes.

```
// DR6 <- 0x00000000
MOV EAX, DR6
AND EAX, 0xFFFFBFFF
MOV DR6, EAX

// EDX:EAX <- LastBranchFromIp
MOV ECX, 0x000001DB; // MSR = 0x01DB(LastBranchFromIp)
RDMSR;

PUSH ES
MOV BX, 0x001B
MOV ES, BX
MOV EDX, EAX
MOV EAX, ES:[EAX]
POP ES

//
// Branch on Instruction type
//

CMP AL, 0xE8            // Relative near call
JZ CALL_FOUND
CMP AL, 0xFF            // Direct near/far call
JZ CALLORJMP_FOUND
CMP AL, 0x9A            // Direct far call
JZ CALL_FOUND

CMP AL, 0xC3            // near RET
JZ RET_FOUND
CMP AL, 0xCB            // far RET
JZ RET_FOUND
CMP AL, 0xC2            // near RET with POP
```

```
JZ RETN_FOUND
CMP AL, 0xCA            // far RET with POP
JZ RETN_FOUND

JMP CALL_NOTFOUND

CALLORJMP_FOUND:
TEST AH, 0x10           // CALL/2
JNZ CALL_FOUND
TEST AH, 0x18           // CALL/3
JNZ CALL_FOUND
CMP AH, 0xE4            // JMP ESP
JZ JMPESP_FOUND

JMP CALL_NOTFOUND
```

4.6. In the case of CALL instruction executes

When execution of CALL instruction is detected in target process, the return addresses are saved in stack recorder according to the following flow chart



① When CALL instruction is executed, the original return address stored in stack area is got according the following codes.

```
CALL_FOUND:
PUSH ES
// Get Stack segment (CS)
MOV ECX, EBP
ADD ECX, + 4 + 4 + 4 + 4 + 4
MOV EAX, [ECX]
MOV ES, AX
// Get Stack pointer
MOV ECX, EBP
ADD ECX, + 4 + 4 + 4 + 4
LES EDX, [ECX]                  // Now EDX point to Stack Address
// Get RetIP
MOV ECX, EDX
MOV AX, 0x001B                  // User mode only
MOV ES, AX                              //
MOV EDX, ES:[ECX]               // Retrieve RetIP on Stack
//
// Now EDX point to RetIP on Stack
//
POP ES
```

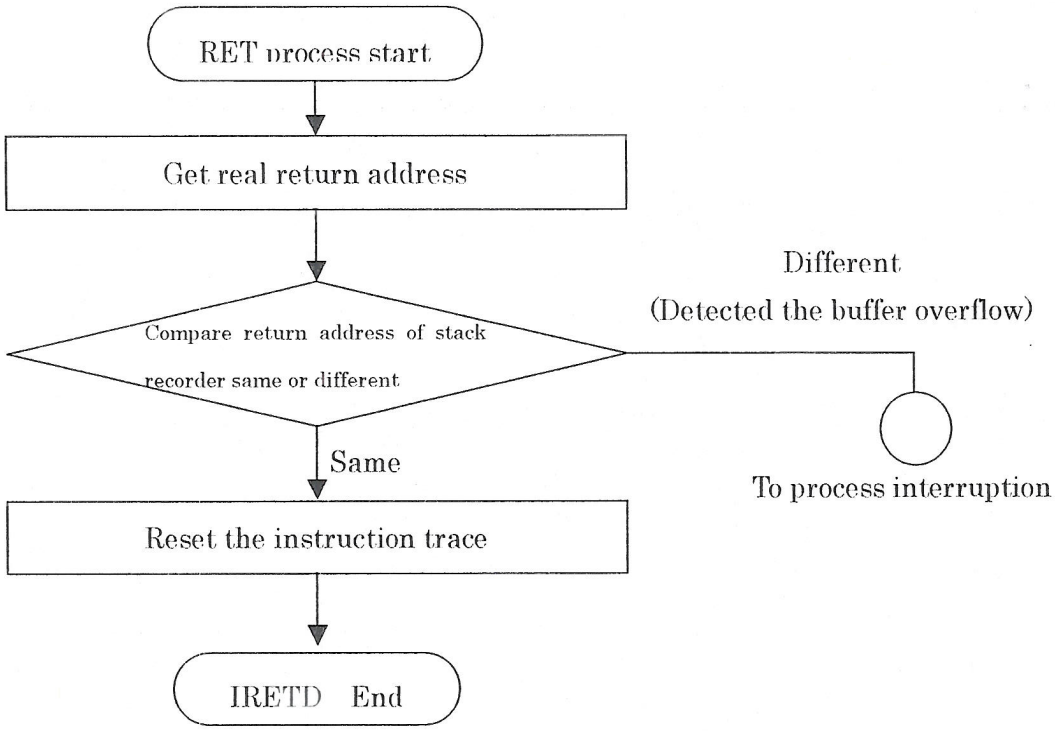
② The return address got from (1) is stored in stack recorder inside the IMMUNE system. It is to be processed as following:

```
KeRaiseIrql(HIGH_LEVEL, &OldIrql);
PASO_STACK_LIST StackList = (PASO_STACK_LIST)gStackList[ThreadId];
if (StackList == 0){
        // Error
}else if (StackList->CurrentStackLocation > STACK_LIMIT){
        StackList = NULL;
}else if (StackList->CurrentStackLocation >= 0){
        StackList->StackPointer[StackList->CurrentStackLocation] = ExpectedRetIp;
        StackList->CurrentStackLocation ++;
}
KeLowerIrql(OldIrql);
```

③ Reset the instruction trace. Because the instruction trace setting is effective until the branch instruction is detected.

4.7. In the case of RET instruction executes

When execution of RET instruction is detected in target process, it is to follow the chart below to check the overflow.



① When RET instruction is executed, the real return address stored in stack area is got according the following codes.

```
RET_FOUND:
PUSH ES
// Get Stack segment (SS)
MOV ECX, EBP
ADD ECX, + 4 + 4 + 4 + 4 + 4
MOV EAX, [ECX]
MOV ES, AX

// Get Stack pointer
MOV ECX, EBP
ADD ECX, + 4 + 4 + 4 + 4
MOV EAX, [ECX]
LES EDX, [ECX]                    // Now EDX point to Stack Address

SUB EDX, +4                       // Back 4Bytes from Current Stack Address
MOV ECX, EDX

MOV AX, 0x001B
MOV ES, AX
MOV EDX, ES:[ECX]
```

② The real return address in stack area and the original return address stored during call instruction executes are compared

```
KeRaiseIrql(HIGH_LEVEL, &OldIrql);
PASO_STACK_LIST StackList = (PASO_STACK_LIST)gStackList[ThreadId];
if (StackList == 0){
        // Stack not found
}else if (StackList->CurrentStackLocation > 0){
        StackList->CurrentStackLocation --;
        ULONG ExpectedRetIp
            = StackList->StackPointer[StackList->CurrentStackLocation];
        StackList->StackPointer[StackList->CurrentStackLocation] = 0;
        if (ExpectedRetIp != ToIp){
                LONG i;
                BOOLEAN StackFound = FALSE;
                for (i = StackList->CurrentStackLocation; i >= 0; i --){
                        if (StackList->StackPointer[i] == ToIp){
                        LONG j;
                                for (j = i; j <= StackList->CurrentStackLocation; j++){
                                        StackList->StackPointer[j] = 0;
                                }
                                StackList->CurrentStackLocation = i;
                                StackFound = TRUE;
                                break;
                        }
                }
                if (!StackFound){
                        // Not found
                        Terminate_VirusCode(FromIp, ToIp, ExpectedRetIp);
                }
        }
}else{
        DbgPrint("   Illegal Stack Location¥n");
}
KeLowerIrql(OldIrql);
```

③ When the return addresses are identical, it is believed to be the normal process and this process is finished. When the addresses are not identical, it is believed that overflow occurs. That is when the process of termination described in 4.7(Terminate VirusCode) will be executed.

④ Reset the instruction trace.

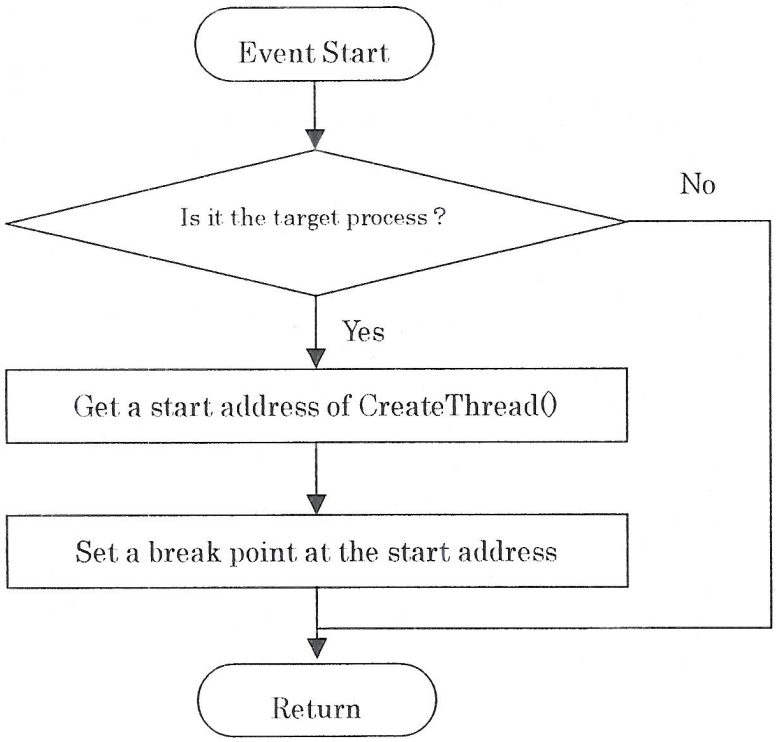
4.8. Force process termination

① The process will be forcefully terminated by the hidden illegal codes in executed instruction address (real return address) according to RET instruction.

```
void __stdcall Terminate_VirusCode(ULONG FromIp, ULONG ToIp)
{
        IsDetected = TRUE;
        // Rewrite FromIp(Next instruction of JMP ESP) to INT3
        __asm{
                PUSH EAX
                PUSH EDX

                MOV AL, 0xCC            // INT 3
                MOV EDX, FromIp
                MOV SS:[EDX], AL

                POP EDX
                POP EAX
        }
}
```

4.9. To set a hook of thread generating events

The sub thread is under monitoring as soon as it is created by target process. So, the system is able to catch the event generated by sub thread.



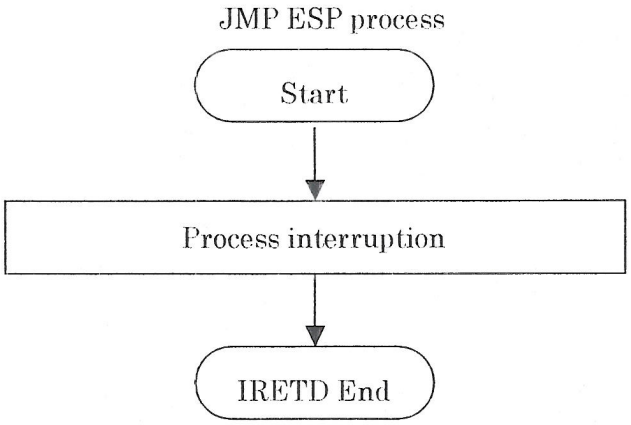
The API Createthread() is hooked when target thread is generated. Whenever sub thread is generated, it starts to monitor the branch instruction same as main thread.

The following codes are used to set the hook of thread generating events (CreateThread()).

```
KIRQL OldIrql;
KeRaiseIrql(HIGH_LEVEL, &OldIrql);

__asm{
        PUSHAD

        // for CreateThread()
        MOV EAX, EBP            // Current EBP
        MOV EAX, [EAX]          // Previous EBP(ASO_Hook_INT2BH)
        MOV EAX, [EAX]          // Previous EBP(CreateThread)
        ADD EAX, 0x10           // Stack + 10H (lpStartAddress)
        MOV EBX, [EAX]          // EBX <- Thread address
        CMP EBX, 0x7800BE4A     // if EBX == _beginthread's start_address (2K+SP0) then

        JNZ SET_MEMORYBREAK

        // for _beginthread()
        MOV EAX, EBP            // 現在の EBP
        MOV EAX, [EAX]          // Previous EBP(ASO_Hook_INT2BH)
        MOV EAX, [EAX]          // Previous EBP(CreateThread)
        MOV EAX, [EAX]          // Previous EBP(_beginthread)
        ADD EAX, 0x0C           // Stack + 0CH (start_address)
        MOV EBX, [EAX]          // EBX <- Thread address

SET_MEMORYBREAK:

        PUSH EBX                        // Param1
        CALL InstallNewInt01Handler
        POPAD
}
```

4.10. About JMP ESP instruction

JMP ESP is necessary instruction to execute codes in stack area, as in the case of virus invaded through network. The program codes in stack area can be invaded through buffer overflow. JMP ESP instruction returns control of program codes. No matter the return address matches or not with the call origin, it will be on the list of prohibition. Because DOS and Windows of early versions use the way to run a program with limited memory, the program codes are created in stack area. In recent years, OS doesn't use this way so it can also be prohibited.



5. BOF test

5.1. BOF sample codes test

We have a simple BOF test that SQL slammer attacking to non-patched Microsoft SQL Server2000 using an emulated program implemented slammer virus code. This emulated program, sqlslammer.exe, force to execute notepad up to infinity when the program attack to non-patched SQL Server. If you set enable for the immune system, the immune system can detect this attack, and immediately kill the process of SQL Server.

5.2. BOF test of metamorphic coding type

Metamorphic coding will carry out the BOF detection. This detection by old way of signal pattern matching was quite hard. The metamorphic coding has generally three types

(1) Register replacement type

```
POP EDX
MOV EDI, 0008H
MOV ESI,EBP
MOV EAX 000DH
ADD EDX, 005FH
MOV EDX,[EDX]
MOV [ESI+EAX*0000CCC9,EBX]

POP EAX
MOV EDX,0008H
MOV EDX,EBP
MOV EDI,000DH
ADD EAX,005FH
MOV ESI,[EAX]
MOV [EDX+EDI*0000CCC9],ESI
```

(2) Magic number exchanging type

```
MOV DWORD PTR [ESI] ,11000000H
MOVDWORD PTR
[ESI+0004],110000FFH
MOV EDI,11000000H
MOV [ESI],EDI
POP EDI
```

```
PUSH EDX
MOV DH,40
MOV EDX,110000FFH
PUSH EBX
MOV EDX,EBX
MOV [ESI+0004],EDX
MOV EDX,11000000H
MOV [ESI],EBX
POP EDX
PUSH ECX
MOV ECX,11000000H
ADD ECX,000000FFH
MOV [ESI+0004],ECX
```

(3) Control order changing type

```
INSTRUCTION_A
INSTRUCTION_B
INSTRUCTION_C:
LABEL_2:
INSTRUCTION B
JMP
FAKE INSTRUCTIONS
LABEL_3:
INSTRUCTION_C
START::
LABEL_1:
INSTRUCTION_A
JMP
FAKE INSTRUCTIONS
```

In the three cases showed above, there is signature change after linkage. IMMUNE system successfully detected BOF and terminated the process since the buffer overflow generated in all the three cases.

6. Extension of IMMUNE system

6.1. For Windows XP machine

This system is suitable for Microsoft Windows 2000. The following points should be taken care when dealing with Windows XP.

① Operation after process detection

In Windows XP, the notice can't be captured even though the breakpoints were set at the very beginning of process. The following changes must be made.

```
__asm{
        PUSHAD
        MOV EAX, CR0;
        PUSH EAX
        AND EAX, ~(0x10000)
        MOV CR0, EAX            // Remove specialized memory flag of reading
}
char *TaleOfImage = (char *)((ULONG)ImageInfo->ImageBase + ImageInfo->ImageSize - 32);

if ((TaleOfImage[5] == (char)0xCD)
&&  (TaleOfImage[6] == (char)0x20)
&&  (TaleOfImage[7] == (char)0xE9)){
        // Already patched
}else{
        // Replace the process
        memcpy(TaleOfImage, EntryPoint, 5);
        TaleOfImage[5] = (char)0xCD;        // INTxx
        TaleOfImage[6] = (char)0x20;        // 0x20
        TaleOfImage[7] = (char)0xE9;        // far JMP
        *(ULONG *)&TaleOfImage[8] = (ULONG)EntryPoint - (ULONG)TaleOfImage - 7;
        EntryPoint[0] = (char)0xE9;         // far JMP
        *(ULONG *)&EntryPoint[1] = (ULONG)TaleOfImage - (ULONG)EntryPoint - 5;
}
```

② Detect the thread generating

The native API call mode of processor supporting SYSENTER/SYSEXIT instruction is changed in Windows XP. So, the old way of INT 2EH hook can't be used any more. Only the native API service table replacement should be used.

```
KIRQL OldIrql;
KeRaiseIrql(HIGH_LEVEL, &OldIrql);

PServiceDescriptorTableEntry Table = &KeServiceDescriptorTable;
OldNtCreateThread = (NTCREATETHREAD)Table->ServiceTableBase[0x35];
Table->ServiceTableBase[0x35] = (unsigned int)NewNtCreateThread;

KeLowerIrql(OldIrql);
```

7. References

1. Ruo Ando, Hideaki Miura, Yoshiyasu Takafuji, "File system driver filtering agensst metamorphic viral coding", WSEAS TRANSACTIONS ON INFORMATIN SCIENCE AND APPLICATIONS. Issue 4, Volume 1, October 2004, ISSN: 1790-0832.

2. Y.TAksfuji, K.shoji, H.Miura, T.Kawade, T.Nozaki, "Security strategy and a proposal of driverware", JNSA 2003.

3. Yoshiyasu Takefuji, "Security Strategy and Management", Nikkei Pub., 2004/10/18, ISBN 486130024X, (Japanese).

Appendix 1 Setup

The component of the module

Name	Remark
AntiStackOverflow.sys	IMMUNE Driver
ASOFace.exe	IMMUNE GUI
IMMUNE.reg	Registry file for starting IMMUNE Driver

Installation

1. Copy IMMUNE driver file

   Copy AntiStackOverflow.sys to the directory of %SystemRoot%¥System32¥drivers

2. Create registry

   Double click IMMUNE.reg to create a registry for starting the IMMUNE driver

3. Enable IMMUNE system

   Restart Windows

4. Customize detecting target process

   It's able to change the name of detecting target process by changing the value of the following registry.

   Key:
   HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥Antistackoverflow¥Parameters
   Value: AppName
   The default value is sqlservr.exe